

# C++ STL best and worst performance features and how to learn from them

Danila Kutenin

Google

[kutdanila@gmail.com](mailto:kutdanila@gmail.com)

Telegram: [@Danlark](#)

Twitter: [@Danlark1](#)

# About myself

- Worked at Yandex on core search engines
- Working at Google on distributed data processing
- Primary expertise is C/C++, low-level design, distributed systems design

# Agenda

- Is C++ about performance?
- Performance problems
  - ABI
  - Compiler
  - Algorithmic
- STL performance experience

# C++ performance

C++ is performant. Because of philosophy [1].

- What you don't use, you don't pay for
- What you do use, you couldn't hand-code any better

[1] B. Stroustrup

<https://dl.acm.org/doi/abs/10.1145/3386320>

# C++ performance

Cppcon | 2019  
the C++ Conference cplusplus.org



Chandler Carruth

There Are No Zero-Cost Abstractions

Video Sponsorship Provided By:  
**ansatz**



Larry & Sergey  
Protobuf Moving Co.  
est. 1998

37

Chandler Carruth “There Are No Zero-cost Abstractions”

# STL (Standard Template Library)

- Provides “standard” things. `std::vector`
- Hard to correctly implement “convenient” things. `std::shared_ptr`

# STL (Standard Template Library)

Is it actually performant?



# STL (Standard Template Library)

Why is `std::regex` so slow?

7:29 PM · Apr 19, 2020 · [Twitter Web App](#)

`std::to_string` faster than `light`

Day 1 / 10:45 / Track 4

C++11 `regex` slower than python

Asked 7 years, 5 months

Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step

Doubling the speed of `std::uniform_int_distribution` in the GNU C++ library

## `std::string` replacement

- We replaced `std::string`'s implementation with `fbstring`'s
- `std::string` and `folly::fbstring` now have implementation, but are still different types

1% performance win

Posted by u/Rseding91 [Factorio Developer](#) 1 year ago

`std::pair<>` disappointing performance

🔗 [Fix PR35637: suboptimal codegen for `vector<unsigned char>`](#)

[rL367183](#)

`libc++` has quadratic `std::sort`

[llvm.org/bugs/s...](#)



# STL (Standard Template Library)

- Results in self made libraries
  - Abseil
  - Folly
  - EASTL
- Why?
  - ABI compatibility and faster progress
  - Speed is simply money

# Outstanding Types/Containers

- `std::array`
- `std::optional`, `std::variant`
- `std::atomic`
- `std::span`, `std::string_view`
- `<algorithm>`

Encouraged to use in many places

# Debatable Types/Containers

- `std::vector`
- `std::string`
- `std::set`, `std::map`

People still debate. Readability costs outweigh the last percentages

# Bad Types/Containers

- `std::pair`, `std::tuple`
- `std::unordered_*`
- `std::regex`

Last two are banned in many places

# `std::array<T>`

- No constructors, copy operators, destructors
  - Rule of zero is the key to success
- The performance is as `T[N]` with the convenient helper functions

# std::array<T>

std::is\_trivially\_copyable if T is

```
1 #include <array>
2 #include <vector>
3
4 void copy(const std::vector<std::array<int, 10>>& v1,
5           std::vector<std::array<int, 10>>& v2) {
6     v2 = v1;
7 }
```

Assembly output (x86\_64):

```
68 testq %r8, %r8
69 je .L11
70 movq %r8, %rdx
71 movq %r15, %rdi
72 movq %r14, %rsi
73 call memmove
74 movq 8(%rbx), %rdi
75 movq (%rbx), %r15
76 movq 8(%rbp), %rdx
77 movq 8(%rbp), %r14
78 movq %rdi, %r8
79 subq %r15, %r8
80 .L11:
81 leaq (%r14,%r8), %rsi
82 cmpq %rdx, %rsi
83 jne .L12
84 .L24:
85 addq %r15, %r12
86 .L8:
87 movq %r12, 8(%rbx)
88 addq $24, %rsp
89 popq %rbx
90 popq %rbp
91 popq %r12
92 popq %r13
93 popq %r14
94 popq %r15
95 ret
96 .L21:
97 ret
98 .L12:
99 subq %rsi, %rdx
100 call memmove
101 addq (%rbx), %r12
102 jmp .L8
103 .L13:
104 xorl %ebp, %ebp
105 jmp .L4
```

Red arrows indicate the mapping from C++ code to assembly:

- From `v2 = v1;` (line 6) to the `call memmove` instruction (line 73).
- From the `std::array<int, 10>>& v2` parameter (line 5) to the `leaq (%r14,%r8), %rsi` instruction (line 81).

# Trick #1

- If possible, make your type trivial
  - Trivially destructible types
    - It allows to “reuse” the object
  - Trivially copyable types can be `memcpy`'ed
    - `mem*` are highly platform optimized

# Trick #1

The SysV ABI specification, section 3.2.3  
Parameter Passing says:

*If a C++ object has either a non-trivial copy constructor or a non-trivial destructor, it is passed by invisible reference.*



# Trick #1

The image shows a C++ source file and its assembly output. The source code defines two classes, `Foo` and `Bar`, each with a default constructor and a member variable `x` initialized to 0. It also defines two copy functions: `foo_cpy` and `bar_cpy`. The assembly output shows the generated code for these functions. An arrow points from the assembly output back to the source code, and a large red "Don't!" is overlaid on the right side of the image.

```
1 class Foo {
2     ~Foo() = default;
3     int x = 0;
4 };
5
6 class Bar {
7     ~Bar() {}
8     int x = 0;
9 };
10
11 void foo_cpy(Foo f1, Foo& f2) {
12     f2 = f1;
13 }
14
15 void bar_cpy(Bar f1, Bar& f2) {
16     f2 = f1;
17 }
18
```

Assembly output (x86-64 gcc (trunk) (Editor #1, Compiler #2) C++):

```
1 foo_cpy(Foo, Foo&):
2     movl %edi, (%rsi)
3     ret
4 bar_cpy(Bar, Bar&):
5     movl (%rdi), %eax
6     movl %eax, (%rsi)
7     ret
```

Don't!

# std::optional<T>



The image shows a screenshot of a C++ development environment. On the left, a window titled 'C++ source #1' displays the following code:

```
1 #include <optional>
2
3 bool f(std::optional<int> t) {
4     return t.has_value();
5 }
```

On the right, a window titled 'x86-64 gcc 10.1 (Editor #1, Compiler #1) C++' shows the assembly output for the function `f`. The compiler options are set to 'x86-64 gcc 10.1' and '-std=c++17 -O3'. The assembly code is as follows:

```
1 f(std::optional<int>):
2     movq    %rdi, %rax
3     shrq    $32, %rax
4     ret
```

# std::optional<T>

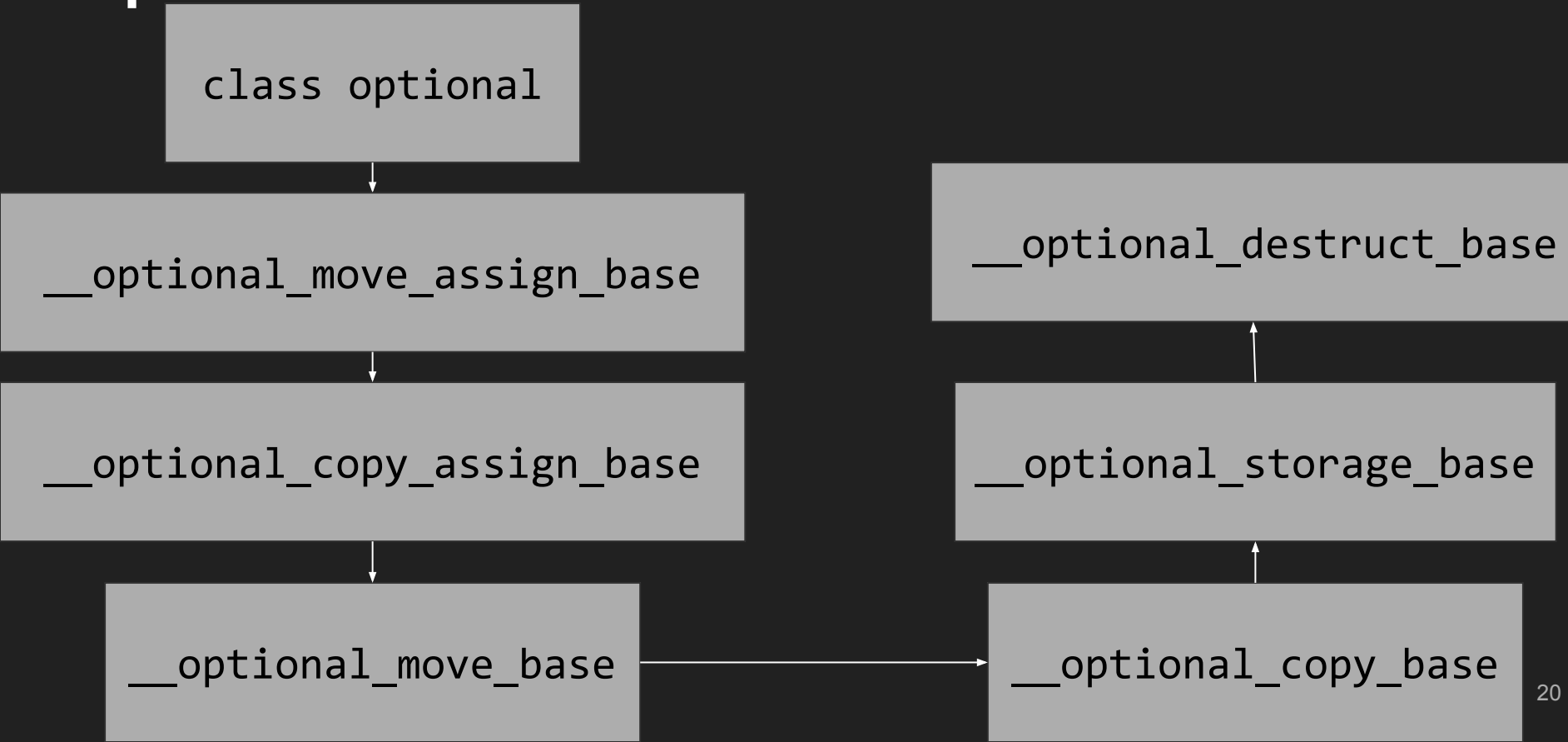
[1]

*~optional();*

1. *#Effects: If `is_trivially_destructible_v<T> != true` and `*this` contains a value, calls `val->T::~~T()`*
2. *#Remarks: If `is_trivially_destructible_v<T>` is true, then this destructor is **trivial**.*

[1] [utilities.optional.destructor](#)

# Optional Perf At What Cost?



# Optional Perf At What Cost?

```
__optional_destruct_base
```

Why?

Partial specializations/SFINAE on special member functions are forbidden

# Optional Perf At What Cost?

1420 Lines of Code\*

\*libc++ implementation

# Optional Perf At What Cost?

P0602R4

variant and optional should propagate  
copy/move triviality

Why is the construction of `std::optional<int>` more expensive than a `std::pair<int, bool>`?

Fixed in libstdc++8

# Evil side. `std::pair`, `std::tuple`

```
1 #include <utility>
2 #include <vector>
3
4 struct MyPair {
5     int a = 0;
6     int b = 0;
7 };
8
9 static void CopyMyPair(benchmark::State& state) {
10     std::vector<MyPair> v1(state.range(0));
11     std::vector<MyPair> v2;
12     for (auto _ : state) {
13         v2 = v1;
14         benchmark::DoNotOptimize(v2);
15     }
16 }
17 BENCHMARK(CopyMyPair) -> Arg(100000);
18
19 static void CopyStdPair(benchmark::State& state) {
20     std::vector<std::pair<int, int>> v1(state.range(0));
21     std::vector<std::pair<int, int>> v2;
22     for (auto _ : state) {
23         v2 = v1;
24         benchmark::DoNotOptimize(v2);
25     }
26 }
27 BENCHMARK(CopyStdPair) -> Arg(100000);
28
```

compiler = clang-9.0 ▾

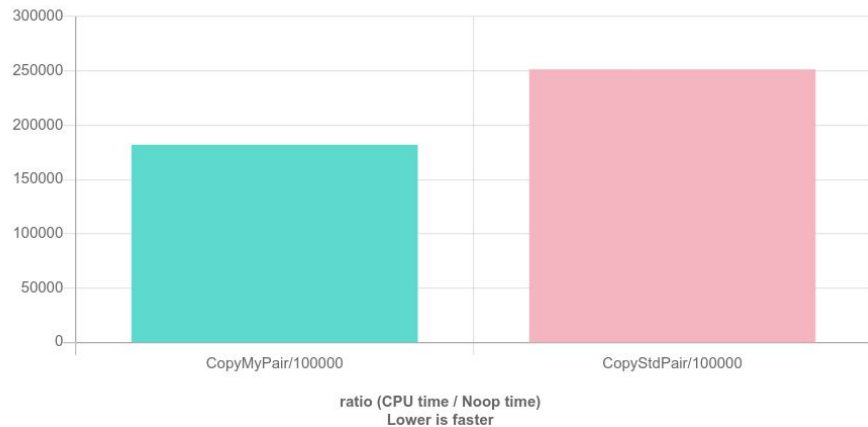
std = c++20 ▾

optim = O3 ▾

STL = libstdc++(GNU) ▾

Run Benchmark

☒ Record disassembly





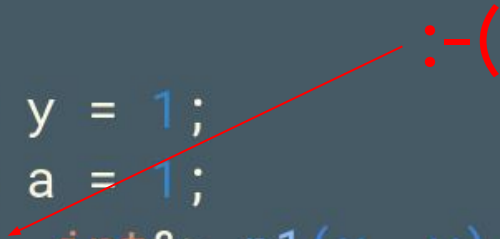
# What?

```
pair& operator=(typename conditional<
                is_copy_assignable<first_type>::value &&
                is_copy_assignable<second_type>::value,
                pair, __nat>::type const& __p)
{
    first = __p.first;
    second = __p.second;
    return *this;
}
```

# Why?

```
#include <iostream>
#include <utility>

int main() {
    int x = 0; int y = 1;
    int z = 0; int a = 1;
    std::pair<int&, int&> p1(x, y);
    std::pair<int&, int&> p2(a, z);
    p1 = p2;
    std::cout << x << ' ' << y << std::endl;
    return 0;
}
```



# Why?

Same with tuple. `std::tie`  
works the same way

# Why?

```
template <class ...Tp>
inline tuple<Tp&...> tie(Tp&... t) noexcept {
    return tuple<Tp&...>(t...);
}
```

# Why?

A defaulted copy assignment operator for class T is defined as deleted if any of the following is true:

- ...
- T has a non-static data member of a reference type;
- ...

# Good news?

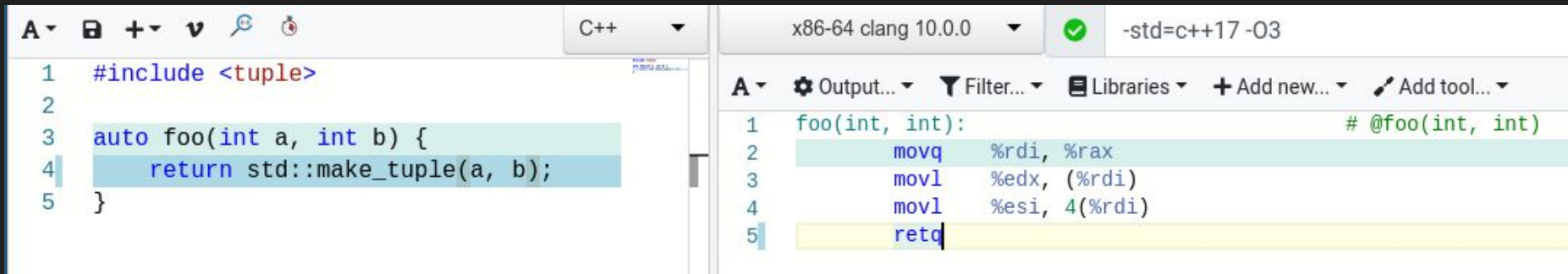
The image shows a C++ IDE with two panels. The left panel displays the source code for a function `foo` that takes two integers and returns a `std::pair`. The right panel shows the assembly output for this function, compiled with `x86_64 clang 10.0.0` using `-std=c++17 -O3`. The assembly consists of five instructions: `foo(int, int):`, `shlq $32, %rsi`, `movl %edi, %eax`, `orq %rsi, %rax`, and `retq`. A red arrow points from the `return` statement in the source code to the `retq` instruction in the assembly, indicating that the function returns a value in a single register.

```
1 #include <utility>
2
3 auto foo(int a, int b) {
4     return std::make_pair(a, b);
5 }
```

```
x86_64 clang 10.0.0 -std=c++17 -O3
1 foo(int, int):                                # @foo(int, int)
2     shlq    $32, %rsi
3     movl    %edi, %eax
4     orq     %rsi, %rax
5     retq
```

Combines in 1 register

# Good news? Not for tuple



The image shows a C++ IDE with two panels. The left panel displays C++ code, and the right panel displays the assembly output for the same code.

**C++ Code (Left Panel):**

```
1 #include <tuple>
2
3 auto foo(int a, int b) {
4     return std::make_tuple(a, b);
5 }
```

**Assembly Output (Right Panel):**

Compiler: x86-64 clang 10.0.0, -std=c++17-O3

```
1 foo(int, int):                                     # @foo(int, int)
2     movq    %rdi, %rax
3     movl    %edx, (%rax)
4     movl    %esi, 4(%rax)
5     retq
```

# Bad news?

## ABI break



# What is ABI?

Translation unit interactions. Including:

- The mangled name for a C++ function
- The mangled name for a type, including templates.
- The number of bytes (sizeof) and the alignment
- The semantics of the bytes in the binary representation of an object.
- Register-level calling conventions.

# What is ABI?

```
struct Example1 {  
    int a = 0;  
    int b = 0;  
    Example1& operator=(const Example1& other) {  
        a = other.a;  
        b = other.b;  
        return *this;  
    }  
};  
  
struct Example2 {  
    int a = 0;  
    int b = 0;  
    Example2& operator=(const Example2& other) = default;  
};
```

# What is ABI?

```
void Example1Copy(const Example1& e,  
                  Example1& e_other) {  
    e_other = e;  
}  
  
void Example2Copy(const Example2& e,  
                  Example2& e_other) {  
    e_other = e;  
}
```

# What is ABI?

```
^^>>> g++ -std=gnu++17 -O0 test_example.cpp -o test_example
^^>>> nm -C test_example | tail
000000000000001200 T __libc_csu_fini
0000000000000011a0 T __libc_csu_init
                  U __libc_start_main@@GLIBC_2.2.5
000000000000001168 T main
0000000000000010a0 t register_tm_clones
000000000000001040 T _start
000000000000004028 D __TMC_END__
000000000000001125 T Example1Copy(Example1 const&, Example1&)
00000000000000114b T Example2Copy(Example2 const&, Example2&)
000000000000001174 W Example1::operator=(Example1 const&)
^^>>> 
```

# Future?

## P0848R3

# Conditionally Trivial Special Member Functions

```
template <typename T>
concept C = /* ... */;

template <typename T>
struct X {
    // #1
    X(X const&) requires C<T> = default;

    // #2
    X(X const& ) { /* ... */ }
};
```

Optional  
implementation  
down to 390  
LOC

## Trick #2

Write =default in your code.  
Always when possible.

# `std::string`

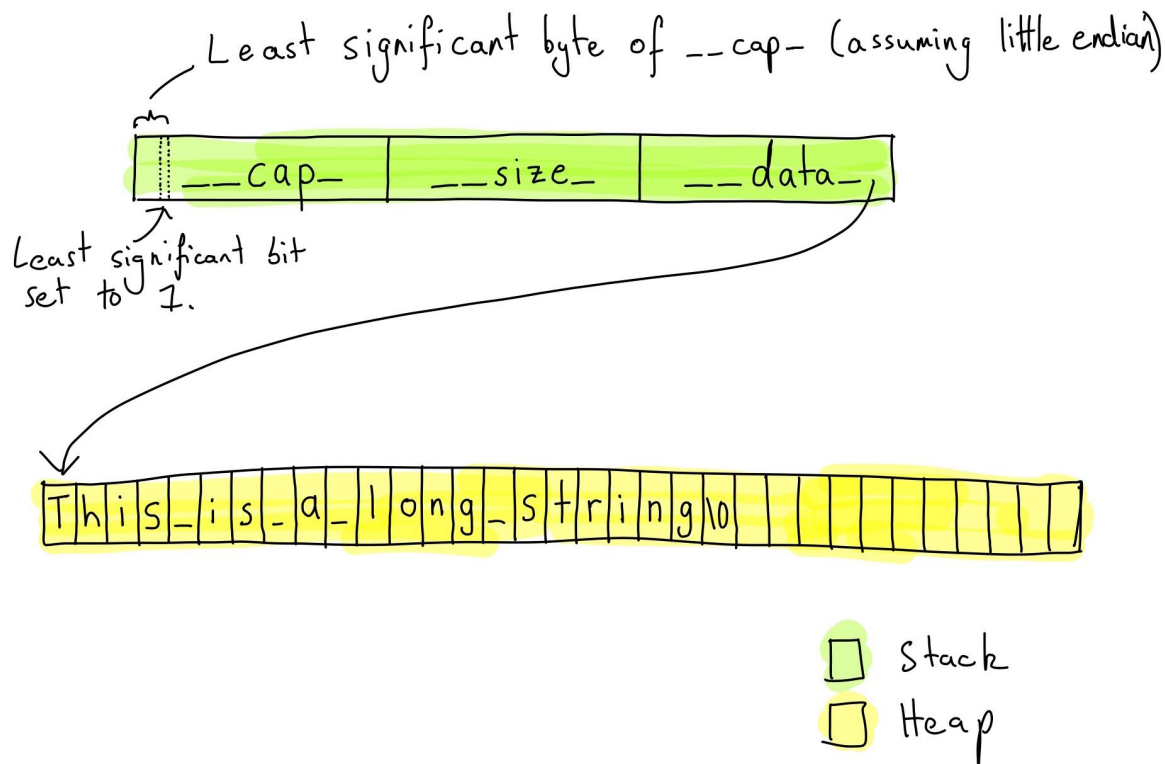
- Small/Short string optimization
  - We must store pointer, size and capacity
  - Reuse these bytes when the string is small
- Dates back to 2000-2001

# std::string

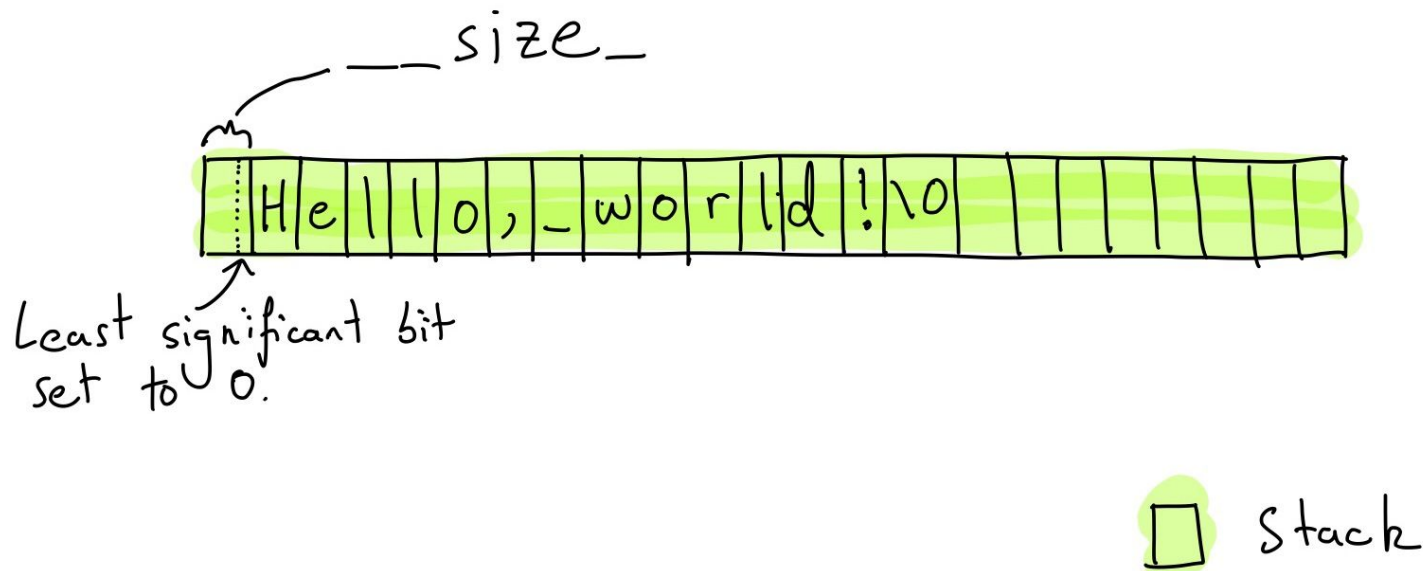
- libc++: 22 bytes
- libstdc++: 15 bytes
- MSVC STL: 15 bytes
- FString: 23 bytes: <https://www.youtube.com/watch?v=kPR8h4-qZdk>
- Yandex: 0 bytes, fully COW



# std::string



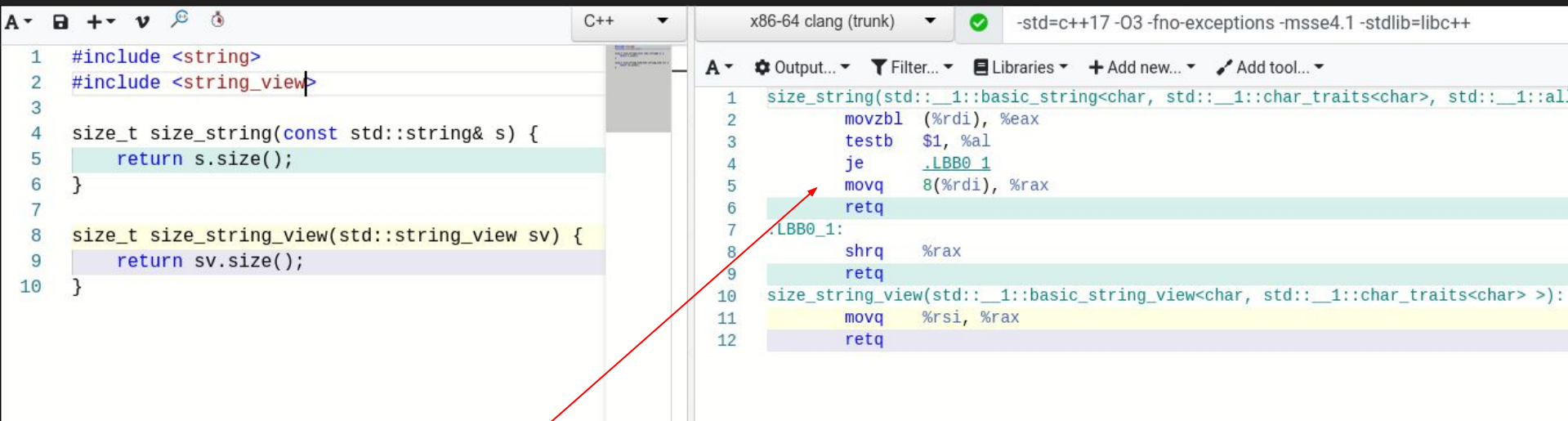
# std::string



# `std::string`

- `std::function` uses the same technique
- The trick can be useful for highly accessed data
- ``const std::string&`` should almost die
  - Use `std::string_view`, two registers, no indirection, cheap copy, pass by value

# std::string



The image shows a C++ IDE with two panels. The left panel displays the source code for `size_string` and `size_string_view` functions. The right panel shows the corresponding assembly code generated by the compiler. A red arrow points from the text 'Check for small string' to the assembly instruction `testb $1, %al` on line 3, which checks if the string length is less than or equal to 1 (a small string).

```
1 #include <string>
2 #include <string_view>
3
4 size_t size_string(const std::string& s) {
5     return s.size();
6 }
7
8 size_t size_string_view(std::string_view sv) {
9     return sv.size();
10 }
```

```
1 size_string(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char>>& s)
2     movzbl (%rdi), %eax
3     testb $1, %al
4     je     .LBB0_1
5     movq  8(%rdi), %rax
6     retq
7 .LBB0_1:
8     shrq  %rax
9     retq
10 size_string_view(std::__1::basic_string_view<char, std::__1::char_traits<char>>& sv)
11     movq  %rsi, %rax
12     retq
```

Check for small string

# std::function

```
std::function<void()> = [&]() {  
    //...  
};
```

# Trick #3

Use `std::string_view` and  
`std::span` almost everywhere

## Trick #4

Remember about small object optimizations, e.g. don't capture blindly by reference in lambda

# std::string

```
std::vector<std::string> to_join;  
std::string result;  
for (const auto& part : to_join) {  
    result += part;  
}
```



# std::string

- absl::StrJoin, folly::join
  - Sums all sizes, does 1 allocation

```
template <typename string_type>
struct ResizeUninitializedTraits<
    string_type,
    absl::void_t<decltype(std::declval<string_type&>()
        .__resize_default_init(237))>>
{
    using HasMember = std::true_type;
    static void Resize(string_type* s, size_t new_size) {
        s->__resize_default_init(new_size);
    }
};
```

# Trick #5

As of C++20, write your own  
string operations library or use  
the existing external one

# `std::unordered_*`

- Check if you need pointer stability (likely not)
- C++17 still does not support heterogeneous lookups
  - Many other libraries do, for example, `absl::flat_hash_*`
- You can outperform `std::` by 10–20x

<https://github.com/google/hashtable-benchmarks>

## Trick #6

If you have enough hash table usages, use external ones or even write your own

# <algorithm>

- Use them, they don't have ABI problems
  - They are constantly optimized in libraries
  - Compilers produce better SIMD code with time
  - Only several are still debatable
    - E.g. `std::sort`, `std::nth_element`
    - Still use them

# `std::rotate`

- Two algorithms, rotating by  $k$  where  $k < n$ .
  - GCD rotate. Moves  $n + \gcd(n, k)$  times.
    - Requires random access
  - Forward rotate. Moves between  $3/2n$  and  $3n$  times.
    - Can be done with forward iterators

# std::rotate

```
template <class ForwardIterator>
inline ForwardIterator
rotate(ForwardIterator first, ForwardIterator middle,
      ForwardIterator last) {
    return rotate(first, middle, last,
                  typename iterator_traits<ForwardIterator>
                      ::iterator_category());
}
```

# std::rotate

```
template <class RandomAccessIterator>
inline RandomAccessIterator
rotate(RandomAccessIterator first, RandomAccessIterator middle,
      RandomAccessIterator last, random_access_iterator_tag) {
    if (is_trivially_move_assignable<value_type>::value) {
        if (next(first) == middle)
            return rotate_left(first, last);
        if (next(middle) == last)
            return rotate_right(first, last);
        return rotate_gcd(first, middle, last);
    }
    return rotate_forward(first, middle, last);
}
```



# std::rotate

```
template <class ForwardIterator>
inline ForwardIterator
rotate(ForwardIterator first, ForwardIterator middle,
      ForwardIterator last, forward_iterator_tag) {
    if (is_trivially_move_assignable<value_type>::value) {
        if (next(first) == middle)
            return rotate_left(first, last);
    }
    return rotate_forward(first, middle, last);
}
```

# std::copy


```
template <class InputIterator, class OutputIterator>  
inline OutputIterator  
copy(InputIterator first, InputIterator last,  
      OutputIterator result) {  
    return __copy(first, last, result);  
}
```

# std::copy

```
template <class InputIterator, class OutputIterator>  
inline OutputIterator  
copy(InputIterator first, InputIterator last,  
      OutputIterator result) {  
    return __copy(first, last, result);  
}
```

# std::copy

```
template <class _Tp, class _Up>
inline
typename enable_if
<
    is_same<typename remove_const<_Tp>::type, _Up>::value &&
    is_trivially_copy_assignable<_Up>::value,
    _Up*
>::type
__copy(_Tp* __first, _Tp* __last, _Up* __result)
{
    const size_t __n = static_cast<size_t>(__last - __first);
    if (__n > 0)
        _VSTD::memmove(__result, __first, __n * sizeof(_Up));
    return __result + __n;
}
```



Bug?

# std::reverse

```
void reverse(const _BidIt _First, const _BidIt _Last) {  
    if constexpr (_Allow_vectorization  
                  && sizeof(_Elem) == 1) {  
        __std_reverse_trivially_swappable_1(_UFirst, _ULast);  
        return;  
    } else if constexpr (_Allow_vectorization  
                          && sizeof(_Elem) == 2) {  
        __std_reverse_trivially_swappable_2(_UFirst, _ULast);  
        return;  
    }  
    ...  
}
```

# std::reverse

```
constexpr bool _Allow_vectorization =  
    conjunction_v<is_pointer<decltype(_UFirst)>,  
                  _Is_trivially_swappable<_Elem>,  
                  negation<is_volatile<_Elem>>>;
```

# std::reverse

```
void __std_reverse_trivially_swappable_2(void* _First, void* _Last) noexcept {
    if (_Byte_length(_First, _Last) >= 64 &&
        _bittest(&__isa_enabled, __ISA_AVAILABLE_AVX2)) {
        const __m256i _Reverse_short_lanes_avx = _mm256_set_epi8( //
            1, 0, 3, 2, 5, 4, 7, 6, 9, 8, 11, 10, 13, 12, 15, 14, //
            1, 0, 3, 2, 5, 4, 7, 6, 9, 8, 11, 10, 13, 12, 15, 14);
        void* _Stop_at = _First;
        _Advance_bytes(_Stop_at, _Byte_length(_First, _Last) >> 6 << 5);
        do {
            _Advance_bytes(_Last, -32);
            const __m256i _Left = _mm256_permute4x64_epi64(
                _mm256_loadu_si256(static_cast<__m256i*>(_First)), 78);
            const __m256i _Right = _mm256_permute4x64_epi64(
                _mm256_loadu_si256(static_cast<__m256i*>(_Last)), 78);
            const __m256i _Left_reversed = _mm256_shuffle_epi8(_Left,
                _Reverse_short_lanes_avx);
            const __m256i _Right_reversed = _mm256_shuffle_epi8(_Right,
                _Reverse_short_lanes_avx);
            _mm256_storeu_si256(static_cast<__m256i*>(_First), _Right_reversed);
            _mm256_storeu_si256(static_cast<__m256i*>(_Last), _Left_reversed);
            _Advance_bytes(_First, 32);
        } while (_First != _Stop_at);
        ...
    }
}
```

# `std::sort`

- Must have  $O(n \log n)$  comparisons
- People debate about the best algorithms
  - pdqsort
  - Introsort
  - countsort
  - etc

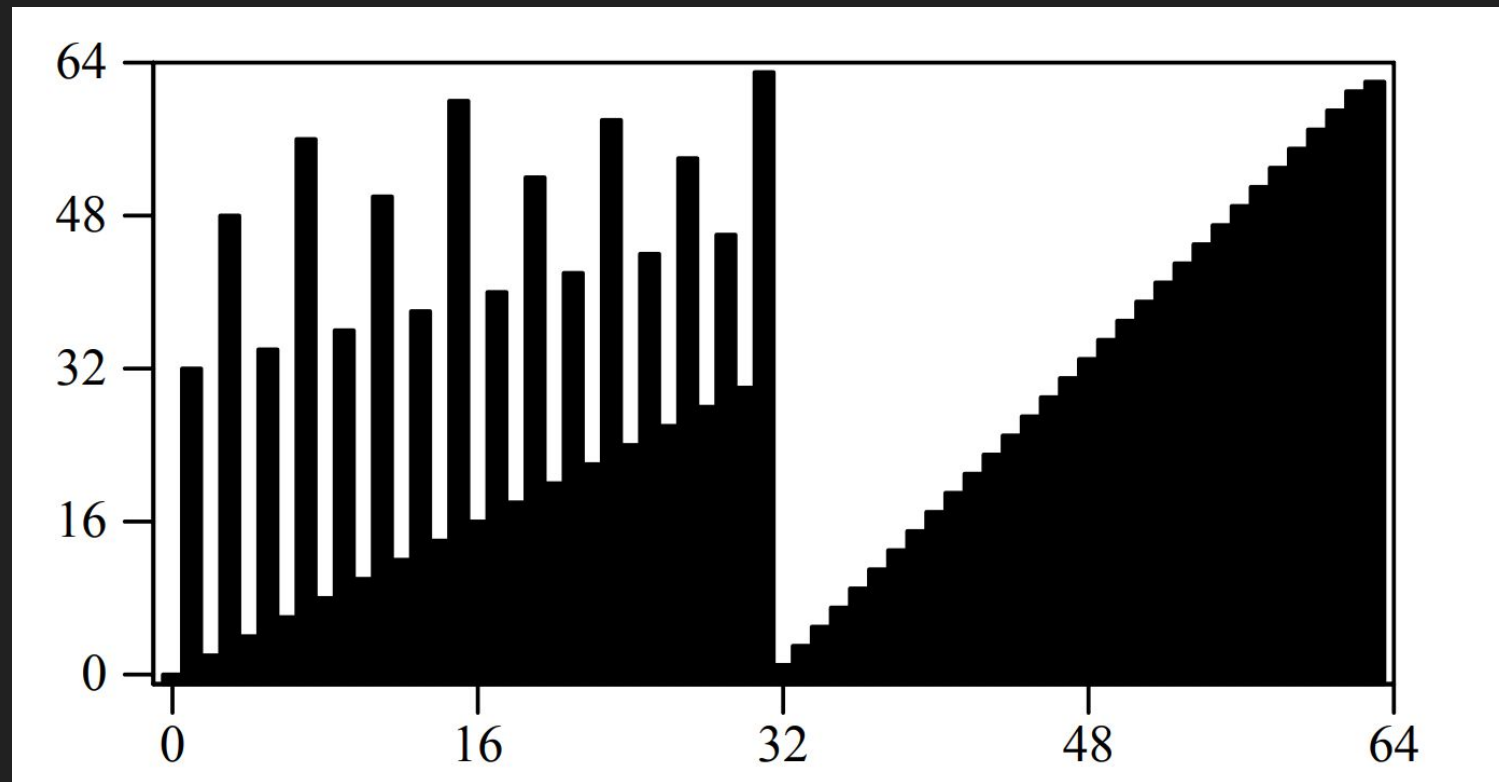


# std::sort

- libc++ has quadratic sort
  - qsort with some tricks

[https://bugs.llvm.org/show\\_bug.cgi?id=20837](https://bugs.llvm.org/show_bug.cgi?id=20837)

# std::sort



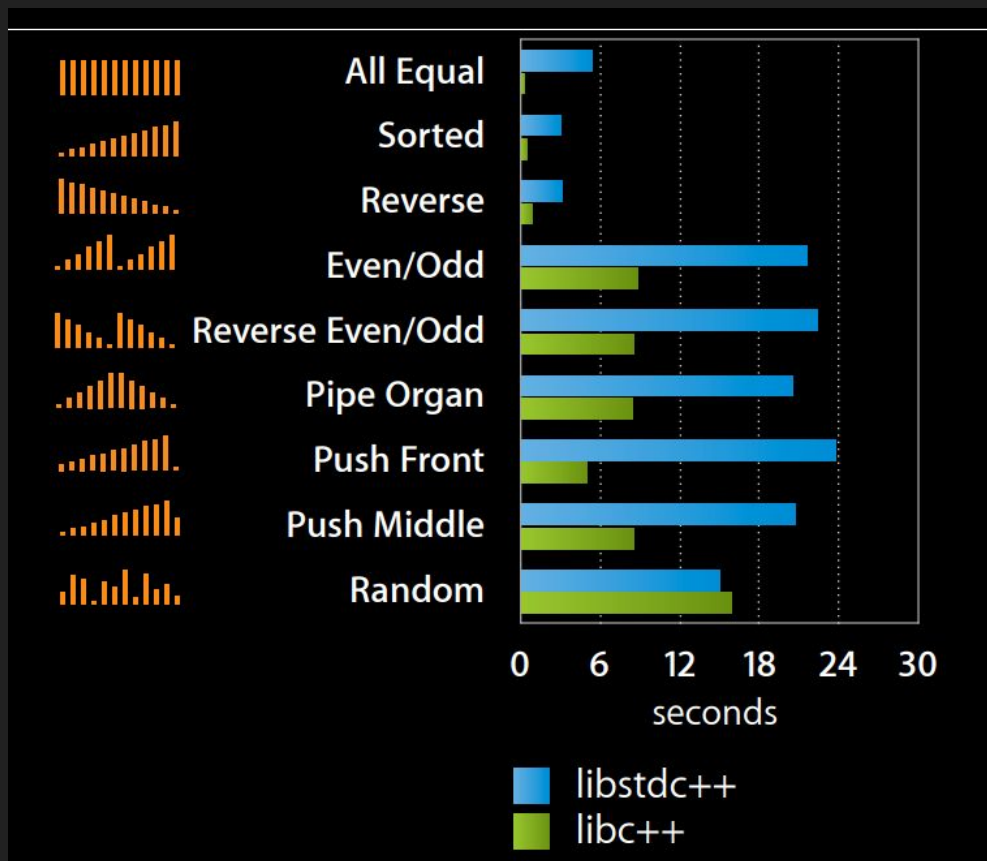
<https://www.cs.dartmouth.edu/~doug/mdmspe.pdf>

# `std::sort`

`n = 1000`

<code>libc++</code>	251232 comparisons
<code>libstdc++</code>	29023 comparisons

# std::sort



# `std::minmax_element`

- Uses  $3/2n + O(1)$  comparisons
  - `min_element` + `max_element` are  $2n$  comparisons
- For trivial types can be worse

# std::minmax\_element

```
1 #include <random>
2 #include <vector>
3 #include <algorithm>
4
5 std::vector<int> generate_data(size_t size) {
6     using value_type = int;
7     std::uniform_int_distribution<value_type> distribution(
8         std::numeric_limits<value_type>::min(),
9         std::numeric_limits<value_type>::max());
10    std::default_random_engine generator;
11
12    std::vector<value_type> data(size);
13    std::generate(data.begin(), data.end(), [&]() { return distribution(generator); });
14    return data;
15 }
16
17 static void MinMaxElement(benchmark::State& state) {
18     auto v = generate_data(100000);
19     for (auto _ : state) {
20         auto [minimum, maximum] = std::minmax_element(v.begin(), v.end());
21         benchmark::DoNotOptimize(minimum);
22         benchmark::DoNotOptimize(maximum);
23     }
24 }
25 BENCHMARK(MinMaxElement);
26
27 static void HandMinMaxElement(benchmark::State& state) {
28     auto v = generate_data(100000);
29     for (auto _ : state) {
30         auto minimum = std::min_element(v.begin(), v.end());
31         auto maximum = std::max_element(v.begin(), v.end());
32         benchmark::DoNotOptimize(minimum);
33         benchmark::DoNotOptimize(maximum);
34     }
35 }
36 BENCHMARK(HandMinMaxElement);
37
```

compiler = Clang 10.0

std = c++20

optim = O3

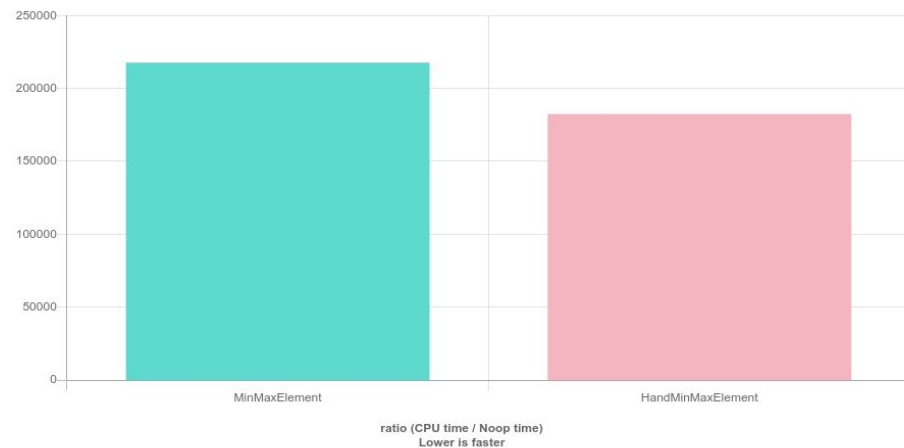
STL = libstdc++(GNU)

Run Benchmark

Record disassembly



Charts Assembly



Show Noop bar

# Trick #7

Use standard algorithms, they  
are almost always good and  
they are constantly improved

# <atomic>

- “Happens before” memory model
- Supported everywhere
  - x86-64, ARM, PowerPC, etc
    - 16 byte atomics!
    - More than 16 is not supported almost anywhere
  - CUDA (finally!)
- `volatile` is deprecated



# <atomic>

Cppcon | 2019  
The C++ Conference  
cppcon.org

JF Bastien

Deprecating volatile

```
struct big { int arr[32]; }  
  
volatile_atomic struct big big_atom;  
struct big b;  
  
void main() {  
    // Yay, this bear!  
    // How many atoms is this atomic "atom"?  
    big_atom = big;  
}
```

16

Video Sponsorship Provided By:  
ansatz

CppCon 2019: JF Bastien “Deprecating volatile”

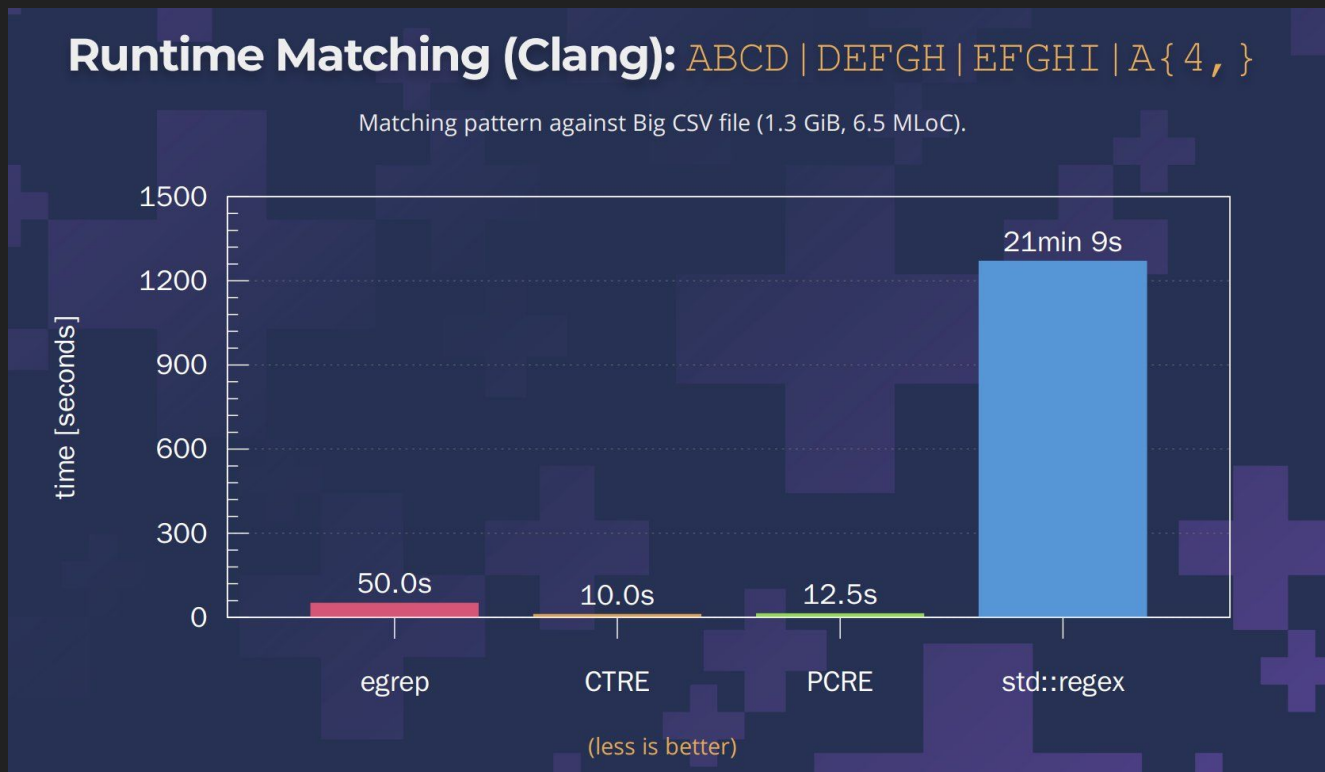
# Trick #8

Use atomics, they are sane

# std::regex

- It must support different grammars
  - BRE, ERE, ECMAScript, grep, egrep, awk, sed, etc.
  - It becomes a part of ABI

# std::regex



# std::regex

- std::regex crashes when matching long lines
  - 2 years
- C++11 std::regex memory corruption
  - 6 years
- C++11 std::regex resource exhaustion
  - 6 years

# Trick #9

Never use `std::regex`

# Trick #9

RE2, Hyperscan, PCRE, CTRE,  
boost::regex are much better

# Real world

- Balance between speed and readability
  - Education, people onboarding
  - Last 1% might be more expensive in a long run
    - Debugging, occasional bugs



# Real world. libc++ vs libstdc++

- Once ClickHouse decided to update the standard library from libstdc++ to libc++

**alexey-milovidov** commented on Dec 22, 2019 • edited

Author Member 😊 ...


And there was unexpected speedup in some queries. For example, formatting in Pretty formants was improved about 40%. Overall performance improvement is about 2%.

😊 1

<https://github.com/ClickHouse/ClickHouse/pull/8311>

# Real world. C++17-20

- C++17-C++20
- char8\_t
- 2% win
- C++14-C++17
- noexcept
- Copy elision
- 99 quantile win

 alexey-milovidov commented on Feb 23 Member

For WITH number AS x SELECT sum(x < 1 ? 1 : (x < 5 ? 2 : 3)) FROM system.numbers  
the difference in generated code looks like this:

```
43b:  mov     0x10(%r13),%rdx
6.06 |      movzbl (%rdx,%rax,1),%ecx
4.55 |      mov     0x10(%rbp),%rdx      !!!
21.21 |      mov     %c1, (%rdx,%rax,1)
1.52 |      add     $0x1,%rax
      cmp     %rax,%r14
1.52 |      je      a80e2b0 <bool DB::FunctionIf::executeTyped<unsigned char, unsigned char>(DB::ColumnVector<
3.03 |      mov     0x10(%r12),%rdx      !!!
10.61 |      cmpb    $0x0, (%rdx,%rax,1)
9.09 |      je      a80e43b <bool DB::FunctionIf::executeTyped<unsigned char, unsigned char>(DB::ColumnVector<
```

vs.

```
456:  mov     0x10(%r12),%rsi
38.46 |      movzbl (%rsi,%rax,1),%esi
21.15 |      mov     %s1l, (%rcx,%rax,1)
      add     $0x1,%rax
      cmp     %rax,%r14
1.92 |      je      197
36.54 |      cmpb    $0x0, (%rdx,%rax,1)
      je      456
```

two extra movs. The difference is clearly attributed to strict aliasing.

# Trick #10

Write benchmarks, try different things, find your own best

# ABI breakages

- [P2028](#) paper about ABI future
  - Prague meeting results:
    - Committee can consider ABI breakage proposals
    - Only for huge performance wins
    - Do not break much
    - Be loud about the decision(?)

# Trick #11

C++ is more than performance

# Questions?